

## CS 5523 Lecture 12: Thread concepts

- Questions on laboratory 2
- Thread state
- Strategies for threading servers
- Kernel-level vs user-level threads
- Posix threads
- Examples

## What is a thread?

- Thread – an abstract data type representing flow of control within a process
- Multiple threads can be created within a single execution environment (a process) to achieve parallelism

## Thread state:

- Thread ID
- Machine state (registers, program counter, stack pointer)
- Priority
- Signal mask
- Error designation (e.g., errno)

## Why threads?

### Upside:

- Cheaper to create than processes
- Cheaper to switch between than processes
- Easier to share data

### Downside:

- Less robust against programming errors
- Have to share resources allocated to the execution environment

Figure 6.5  
Client and server with threads

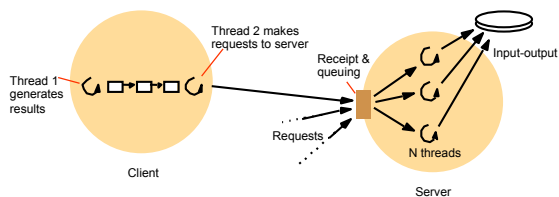
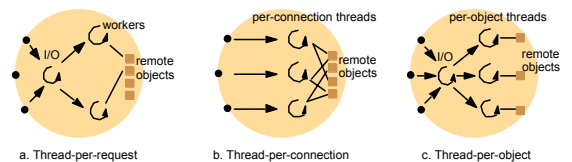


Figure 6.6  
Alternative server threading architectures (see also Figure 6.5)



## Thread operations:

- **Thread management:**
  - creation
  - destruction
  - changing priority or other attributes
- **Synchronization**
  - join
  - condition variables
  - locks

## Kernel-level threads:

- Each thread is a schedulable entity
- Threads compete for process resources on system-wide basis
- Can take advantage of multiple processor
- Operating system must provide support
- Context switch involves the kernel and can be expensive

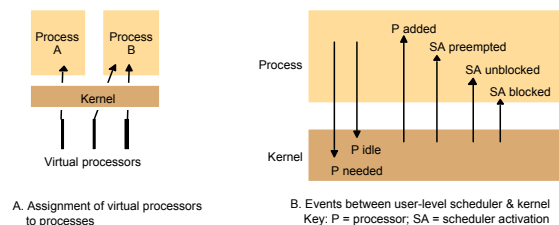
## User-level threads:

- Run on top of existing operating systems
- Encapsulated with processes and invisible to kernel
- Schedule by a runtime system that is part of process code
- Each library function and system call is enclosed by a jacket to let the run-time library *i*

## Hybrid approaches:

- Application contains user-level scheduler to manage threads within a process
- Kernel allocates virtual processors for a process
- Kernel makes upcalls to notify application's thread scheduler of events:
  - Virtual processor allocated
  - Scheduler activation blocked
  - Scheduler activation unblocked
  - Scheduler application preempted

Figure 6.10  
Scheduler activations



## Examples of different approaches:

- User-level threads – *pthreads*
- Kernel-level threads – *NT, Solaris Lightweight Processes*
- Hybrid approaches – *Solaris threads + LWP FastThreads*

*Java threads are user-level, however, their mapping to underlying thread support is implementation dependent. (Threads are scheduled differently on different underlying OS's.)*

## Posix threads:

- Standardized in 1995 as part of the Posix.1c standard
- User-level
- Parameters are specified by created attribute objects
- All calls start with `pthread_`
- Calls return 0 if successful or a nonzero positive value for an error

## Posix thread management:

- `pthread_create` – create a new thread
- `pthread_join` – wait for a particular thread to terminate
- `pthread_self` – find out own thread ID
- `pthread_detach` – set thread to release resources when it terminates
- `pthread_exit` – terminate self
- `pthread_kill` – terminate another thread

## Creating a Posix thread:

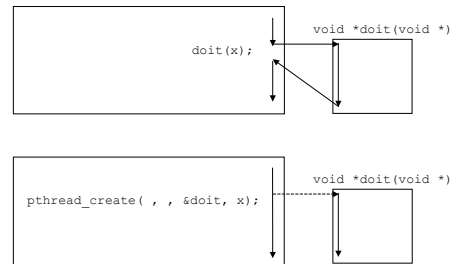
```
cc - mt [ flag... ] file...- lpthread [ library... ]
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void *), void *arg);
```

### Example:

```
{
pthread_t tid;
int fd;
fd = open("aFile.dat", O_RDONLY);
pthread_create(&tid, NULL, &doit, (void *)fd);
...
}

void *doit (void *arg) /* passing an integer argument by value */
{
/* ... process data from file */
close((int)arg);
}
```

## Thread versus function call:



## Joining with a thread

```
int pthread_join(pthread_t thread, void **value_ptr);
```

### Example of joining:

```
{
pthread_t tid;
int fd;
fd = open("aFile.dat", O_RDONLY);
pthread_create(&tid, NULL, &doit, (void *)fd);
pthread_join(tid, NULL);
}

void *doit (void *arg)
{
/* ... process data from file */
close((int)arg);
}
```

You must know the thread ID to join with it. What implications does this have for servers? What if you don't join?

## Detaching a thread

```
int pthread_detach(pthread_t thread);
```

### Example of detaching:

```
{
pthread_t tid;
int fd;
conffd = open("aFile.dat", O_RDONLY);
pthread_create(&tid, NULL, &donowait, (void *)fd);
}

void *donowait (void *arg) {
pthread_detach(pthread_self());
/* ... process data from file */
close((int)arg);
}
```

## Incorrect parameter passing for multiple threads:

```
{
    pthread_t tid;
    int fd;
    int i;
    char buf[1024];
    for (i = 0; i < 20; i++){
        sprintf(buf, "aFile%d.dat", i);
        fd = open(buf, O_RDONLY);
        pthread_create(&tid, NULL, &doit, (void *)fd);
    }
}

void *doit (void *arg) {
    pthread_detach(pthread_self());
    /* ... process data from file */
    close((int)arg);
}
```

*Why won't this code work?*

## Correct parameter passing for multiple threads:

```
{
    pthread_t tid;
    int *fdp;
    int i;
    char buf[1024];
    for (i = 0; i < 20; i++){
        sprintf(buf, "aFile%d.dat", i);
        fdp = malloc(sizeof(int));
        *fdp = open(buf, O_RDONLY);
        pthread_create(&tid, NULL, &doitp, (void *)fdp);
    }
}

void *doitp (void *arg) {
    int theFile = *(int *)arg;
    free(arg);
    pthread_detach(pthread_self());

    /* ... process data from file */
    close(theFile);
}
```

## Programming issues:

- You **MUST** catch errors on **EVERY** thread call:

```
if (error = pthread_create(&tid, NULL, doit, (void *)fd))
    fprintf(stderr, "Couldn't create: %s\n", strerror(error));
```

*Why can't you use perror?*

- You **MUST** check every other system call to make sure that it is thread-safe. (See the man pages for each call.)
- Stevens hides the error handling issue by replacing all system calls to calls to jacket functions that handle the errors. The jacket functions have the same name, but start with a capital letter. For example he uses `Pthread_create` instead of `pthread_create`.

## For next time:

- Read Stevens / Chapter 23.4 - 23.9
- Read the Mars Pathfinder article
- Work on CDK questions 6.4, 6.8, 6.9, 6.10, 6.14, 6.23, 6.24, 6.25